Escuela Politécnica Superior

Unsupervised Learning for Domain Adaptation in automatic classification tasks through Neural Networks

Degree in Computer Engineering

Final Degree Project

Author:
Tudor Nicolae Mateiu

Tutor/s:
Antonio Javier Gallego
Jorge Calvo Zaragoza

Universitat d'Alacant
Universidad de Alicante

September 2019

Unsupervised Learning for Domain Adaptation in automatic classification tasks through Neural Networks

Author
Tudor Nicolae Mateiu (Student)

Directors
Antonio Javier Gallego (Tutor)
Departamento de Lenguajes y Sistemas Informáticos
Jorge Calvo Zaragoza (Co-tutor)
Departamento de Lenguajes y Sistemas Informáticos

Grado en Ingeniería Informática

Escuela
Politécnica
Superior

ALICANTE, September 1, 2019

Abstract

Machine Learning systems have improved dramatically in recent years for automatic recognition and artificial intelligence tasks. In general, these systems are based on the use of a large amount of labeled data - also called training sets - in order to learn a model that fits the problem in question. The training set consists of examples of possible inputs to the system and the output expected from them.

Achieving this training set is the main limitation to use Machine Learning systems, since it requires human effort to find and map possible inputs with their corresponding outputs. The situation is often frustrating since systems learn to solve the task for a specific domain - that is, a type of input with relatively homogeneous conditions - and they are not able to generalize to correctly solve the same task in other domains.

This project considers the use of Domain Adaptation algorithms, which are capable of learning to adapt a Machine Learning model to work in an unknown domain based on only unlabeled data (unsupervised learning). This facilitates the transfer of systems to new domains because obtaining unlabeled data is relatively cheap, since the cost is to label them. To date, Domain Adaptation algorithms have been used in very restricted contexts, so this project aims to make an empirical evaluation of these algorithms in a greater number of cases, as well as propose possible improvements.

Acknowledgments

First of all, I would like to dedicate this project to my parents, without whom none of it could have been possible. They have kindly supported me and helped me reach this point in my life, all thanks to their hard work and arduous belief in me.

Additionally, I would like to express my most sincere gratitude to my two tutors, Antonio Javier Gallego and Jorge Calvo Zaragoza, for everything they've done regarding this project and the incredible opportunity they've given me of writing my first ever scientific article.

We live in a society exquisitely dependent on science and technology,
in which hardly anyone knows anything about science and technology.

Carl Sagan.

We live on this speck called Earth - think about what you might do,
today or tomorrow - and make the most of it.

Neil deGrasse Tyson.

Contents

List of Figures

Table Index

Listing Index

1 Introduction

Machine learning is a branch of Artificial Intelligence tasked with the scientific study and development of algorithms that can "learn" from data provided to them. Unlike traditional Artificial Intelligence algorithms, where the behaviour is explicitly dictated by programming rules written by people in order for it to perform a task, Machine Learning algorithms build a mathematical model, or hypothesis, from a training dataset, and their knowledge is then used for predictions or decisions given a series of future inputs.

Machine Learning has various methods of learning or building the mathematical model needed for the task at hand [1], some of the most used ones are:

- Supervised Learning. Where input and output (labeled) training data is available and the algorithm must learn the mapping function from the input to the output. The goal is to approximate the hypothesis so that for new inputs the algorithm can predict their output (label) correctly.

- Unsupervised Learning. Where only input training data is available and no corresponding output data. The goal for unsupervised learning is to model the underlying structure or distribution in the data in order to learn more about it.

- Semi-supervised Learning. Where there is a large amount of input training data and only some of the data is labeled. Many real world Machine Learning problems are part of this type of learning, because labelling data is a costly task. It is a combination of Supervised and Unsupervised Learning.

Typically, Deep Learning is used, a more modern technique, with its own benefits, encompassed within Machine Learning. The use of Deep Learning algorithms is carried out thanks to complex architectures called Artificial Neural Networks, also known as Deep Neural Networks or Neural Networks (NN) for short, which are loosely inspired by the biological neural networks that constitute an animal's brain. These NN conduct the learning process and are made up of an input layer, one or several hidden layers, and the output layer. The layers in turn consist of connected Artificial Neurons that compute input data and send output data to other connected neurons. Like this, data that enters the input layer, gets passed and processed between layers until it reaches the output layer, depending on the type of NN this can be done several times and by

different methods. The most famous type of these networks are the ones referred to as Convolutional Neural Networks (CNN), commonly used for image classification tasks.

As it has already been mentioned, Machine Learning algorithms, and thus Deep Learning algorithms as well, learn from input data. This may vary from problem to problem as there are many types. This project will deal with pattern recognition using NN, and the data fed to the algorithm will be in the form of images.

1.1 Motivation

Although these NN are somewhat based on how the human brain works and how humans learn and apply their knowledge, a problem can arise because they are limited by the data available for the training task [2, 3]. This problem can occur depending on the next two reasons we will study in this project.

The first has to do with the amount of data some NN need in order to learn a model and apply it to deal with a certain task. As some of these are required to be trained on massive amounts of labeled data, there are two options that can be carried out: one is to label new data, but this is a monetarily costly and time consuming task, and the other is to generate synthetic data, which may be created in abundance but can inevitably suffer from a domain shift (differences in domain between the data) that will negatively influence the training performance.

The second has to do with the domain to which the data belongs to. This can negatively influence the complexity of the classification task because most Machine Learning models fail to transfer their knowledge from one domain to another, thereby requiring to learn from scratch on new domains after manually labeling new data. As mentioned in the first reason, given that on most occasions NN require large amounts of data to learn a model, learning everything again from scratch is an unfeasible and unpractical option. A practical example of this can be: after having a person learned to play the classical guitar, one can safely assume they could successfully apply this knowledge to play the electric guitar, even if only partially by still having a good advantage over starting to learn to play it from a beginner level.

And so, this project's aim is to study this problem through the research and understanding of Domain Adaptation, and the implementation of a specialized type of Neural Network, called Domain Adversarial Neural Network (DANN), that can learn from data, ignore the domain from where the data originates and transfer the knowledge to a new domain [3]. All of this will be tested and experimented on by classifying images from the MNIST dataset and a Mensural Notation Manuscript (MNM) dataset.

1.2 Objectives

Different essential tasks were carried in order to complete the project's aim, from the study and collecting of information about the techniques and tools to be used, to the implementation of algorithms and additional functions needed, and finally to the preparation and interpretation of the data employed and the results obtained. The objectives, in further detail, are:

- **Study and implementation of a Domain Adversarial Neural Network.** The technique of Domain Adaptation will be researched and explained in order to understand why the need to build a unique model and subsequently carrying out the particular implementation, preparation and training process the DANN model requires.

- **Study and implementation of a standard Convolutional Neural Network.** A CNN model trained with standard backpropagation will be implemented to ultimately compare results with the DANN model, and its basic theory and function will be researched and explained in order to understand the differences between it and the Domain Adaptation approach.

- **Preparation and processing of the datasets.** The MNIST and MNM datasets need to be prepared before starting any of the experiments. This means structuring and organizing, via functions that need to be implemented, the datasets in distinct and easily accessible domain groups so that they can be correctly used for the experimentation phase.

- **Comparison and interpretation of experiments of the DANN and CNN approaches.** Both models will be trained and evaluated on a source domain, but the DANN model will additionally be trained on a target domain. The results will be compiled and, afterwards, interpreted.

Given the scientific scope of this project, the most important objectives have not so much to do, for example, with the implementation of the code used, but with the research of the State of the Art techniques and algorithms, and the comparison and understanding of the results obtained by these techniques.

1.3 Project structure

To facilitate its reading, the content of this document is organized into chapters and sections. The structure followed is described below.

- ***Chapter 1: Introduction*** ⇒ Introductory chapter that covers the fundamentals of the project and describes the objectives that are intended to be achieved.

- ***Chapter 2: State of the Art*** ⇒ Explanatory chapter of the techniques and algorithms that are either employed throughout the project or are needed as a basis in order to understand the work carried out.

- ***Chapter 3: Technologies*** ⇒ Chapter containing an explained enumeration of all the technologies and tools utilized to complete the project.

- ***Chapter 4: Methodology*** ⇒ Chapter that deals with the systematic, theoretical analysis of the body of methods and principles applied in this project.

- ***Chapter 5: Implementation*** ⇒ Chapter explaining the realization of the technical specifications, algorithms and functions required to carry out the experimentation stage.

- ***Chapter 6: Experimentation*** ⇒ Chapter detailing the set of reflections obtained regarding the study of the results of the tests applied.

- ***Chapter 7: Conclusions*** ⇒ Chapter containing a summary of the methodology followed, a summary of the conclusions drawn from the experiments and some possible future improvements.

2 State of the art

2.1 Supervised Learning Algorithms

Supervised Learning Algorithms [1] make use of a fully labeled dataset that contains inputs which have a class, or label, associated to them, to learn a function to map these inputs to their label. For example, the images in the MNIST dataset are of numbers, and they are annotated with the corresponding digit present in the image. A Supervised Learning Algorithm can study the MNIST dataset and learn features that enables it to classify images of numbers into ten different classes, the digits from zero to nine. This example is what is called an image classification task.

For Supervised Learning Classification Algorithms given:

- X, the input space.
- Y, the output space (or label space).
- D_S, a source domain over $X \times Y$.

A labeled sample $S = \{(x_i, y_i)\}_{i=1}^{N} \sim (D_S)^N$ is drawn $i.i.d$ from D_S, where N is the total number of samples.

The objective for these classification algorithms is to learn the mathematical model, or hypothesis,

$$h : X \to Y$$

from S so that new examples from D_S are predicted with as little error as possible, thus building what is known as a classifier in many NN, including in Convolutional Neural Networks.

2.2 Convolutional Neural Networks

Convolutional Neural Networks [4] are a specialized kind of NN for processing data that has a known grid-like topology and are most commonly applied to analyzing visual imagery. They make use of a mathematical linear operation called convolution at least once throughout its architecture.

Its design is comprised of connected layers that are made up of a group of Artificial Neurons. Neurons from one layer receive input from a specific group of neurons of the previous layer, this group is called a receptive field. The architecture has an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of a series of convolutional layers that convolve with a multiplication or other scalar product.

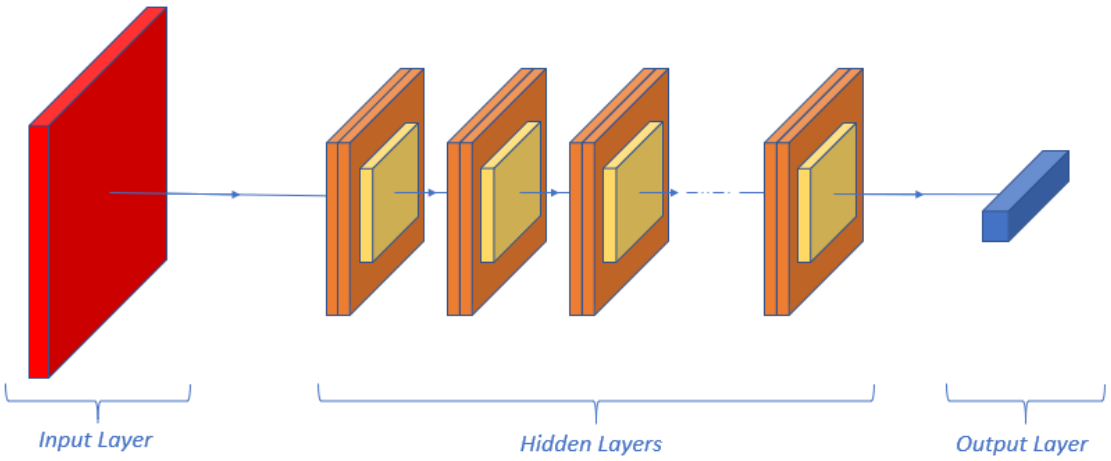

Figure 2.1: Diagram of a CNN architecture with an unspecified amount of hidden layers.

During the training process, the hidden layers of the network are the ones responsible for the extraction of features that will be used in order to classify new inputs. Because of this, the body of hidden layers can also be called the Feature Extractor and the final layer(s), including the output layer, the Label Classifier.

Figure 2.2: Basic CNN classification model diagram. Where x_i is the input and y_i its label.

In turn, an Artificial Neuron [5] is composed of a set of weighted inputs, so for any given one, let there be $n+1$ inputs, x_1 through x_n with corresponding weights w_1 through w_n. The input x_0 is called the bias and has the weight w_0 associated to it. In the final layers of the network, these inputs are passed through a transfer function and afterwards an activation function; thus, each neuron computes its output value by applying these two to the input values coming from the receptive field in the previous layer.

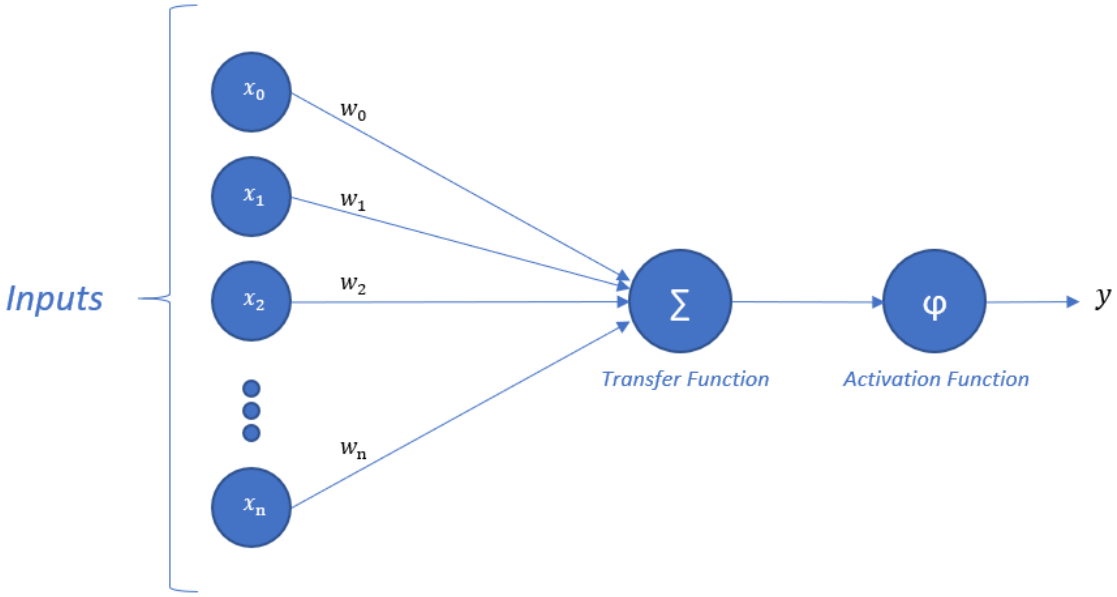

Figure 2.3: Diagram of an Artificial Neuron.

The main point that must be retained from the previous definition of what constitutes a CNN, is that its elemental units, the Artificial Neurons, are made up of a set of inputs with corresponding weights, and that these weights are what is referred to as the extracted features or knowledge of the network.

Because these models send input from an initial layer, through all the hidden layers and finally through the output layer, they are known as feedforward networks. But, for training, a technique called backpropagation is used, which will consist in gradually adjusting the weights of the network based on the error made in the predictions made with the training data.

2.2.1 Backpropagation

Backpropagation algorithms [6] are a family of methods used to efficiently train NN's following a gradient descent approach that exploits the formula for computing the deriva-

tive of the composition of two or more functions.

The main feature of backpropagation is that it enables feedforward models to become iterative, recursive and efficient at calculating the weights updates to improve the network until it is able to perform the task for which it is being trained.

Before training begins, the weights of the network are all set randomly (although there exist techniques that propose better weight initialization). Then, each neuron essentially learns from training examples, which in this case consist of a set of tuples of inputs and corresponding correct outputs, (x_i, t_i). Initially, the network will compute an output, y_i, that likely differs from the correct one, t_i, given that the initial weights of the network are random.

From this, a loss function $L(t, y)$ is used for measuring the discrepancy, or error, E between the expected output and the actual output. In a way, the training process can be conceptualized as the need to produce an output that exactly matches the expected output, which means the error will be zero. In other words, learning can be thought of as an optimization problem to find a hypothesis that minimizes the error.

Because the model's output is obtained by the values of the neurons' weights, so also the error depends on these, therefore they are ultimately what needs to be changed in the network to enable learning. In our case, together with backpropagation, the gradient descent algorithm is used to find the set of weights that minimizes the error.

The gradient descent method involves calculating the derivative of the loss function with respect to the weights of the network in order to update the weights accordingly. In the case of CNN's, the partial derivative is calculated. Given the following:

- $E = L(t, y)$, the error from the loss function given two outputs.
- t, the target output for a training sample.
- y, the actual output of the output neuron for the training sample of t.
- θ_{ij}, the weight j of neuron i.
- lr, the learning rate, this controls the "step size" by which weights are updated.

The partial derivative is calculated as:

$$\frac{\partial E}{\partial \theta_{ij}}$$

Afterwards, the gradient descent method updates each weight according to the subtraction of the weight's original value and the multiplication of the learning rate and partial derivative, the following is the equation used to update the weights:

$$\theta_{ij} = \theta_{ij} - (lr \times \frac{\partial E}{\partial \theta_{ij}})$$

Backpropagation does this for each of the weights and for all the examples in the training dataset. Every pass over all the examples is called an epoch.

2.3 Domain Adaptation

The Domain Adaptation scenario [2, 3] arises when we aim at learning from a source data distribution (domain) a well performing model and applying the knowledge on a different, but related, target data distribution. The appeal of the Domain Adaptation approaches is the ability to learn a mapping between domains in the situation when the target domain data is either fully unlabeled (Unsupervised Domain Adaptation) or have few labeled samples (Semi-supervised Domain Adaptation).

This project focuses on Unsupervised Domain Adaptation, and makes partial use of Supervised Learning Algorithm techniques. Given:

- X, the input space.
- Y_L, the label output space.
- Y_D, the domain output space.
- D_S, a source domain over $X \times Y_L$.
- D_T, a target domain over $X \times Y_L$.

An Unsupervised Domain Adaptation learning algorithm is then provided with a labeled source sample S drawn *i.i.d.* from D_S, and an unlabeled target sample T drawn *i.i.d.* from D_T^X, where $(D_T^X)^{n'}$ is the marginal distribution of D_T over X.

$$S = \{(x_i, y_i)\}_{i=1}^{n} \sim (D_S)^n; \;\; T = \{x_i\}_{i=n+1}^{N} \sim (D_T^X)^{n'},$$

with $N = n + n'$ being the total number of inputs.

The goal of the learning algorithm is to build a label and a domain classifier,

$$h : X \to Y_L; \quad \eta : X \to Y_D,$$

so that the final classification decisions are made based on features that are both discriminative and invariant to the change of domains, theoretically enabling the transfer of knowledge from a source domain to a target domain.

Because of these specifications, a Supervised Learning CNN approach is not appropriate and the design of a special type of NN must be carried out.

2.3.1 Domain Adversarial Neural Network

The architecture experimented on, the Domain Adversarial Neural Network (DANN), was firstly proposed in the work of [3]. It consists of three parts, two of which are common in any standard feed-forward CNN model: the *feature extractor* and *label classifier* (or *label predictor*), as seen in Fig. 2.2.

In order for the classification decisions of the label classifier to be made based on features that are both discriminative and invariant to the change of domains, a domain classifier, which contains a Gradient Reversal Layer, must be added to the model.

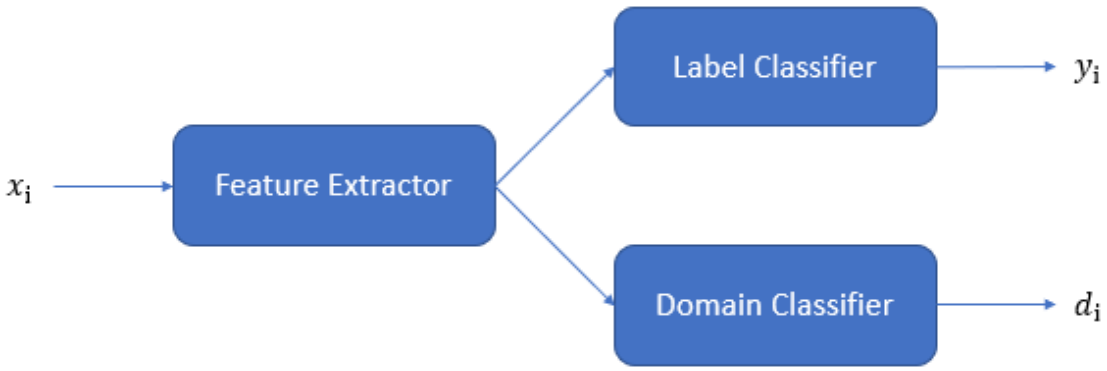

Figure 2.4: Basic DANN classification model diagram. Where x_i is the input, y_i its label and d_i the domain it belongs to.

The Gradient Reversal Layer from the domain classifier multiplies the feature extractor's gradient by a specified negative value during the Backpropagation training. So that the partial gradient is calculated as:

$$-\lambda \frac{\partial E}{\partial \theta_{ij}}$$

And the equation for updating the weights during Backpropagation changes to:

$$\theta_{ij} = \theta_{ij} - (lr \times -\lambda \frac{\partial E}{\partial \theta_{ij}})$$

This prevents the training to be performed in a standard way and will carry out what we seek: to maximize the loss of the domain classifier, which in turn ensures domain invariant features to emerge, disabling the model from learning which domain the input belongs to.

These classifiers will work in tandem so that the label classifier is optimized to *minimize* the loss of label predictions, and the domain classifier is optimized to *maximize* the loss of the current domain prediction, assuring that a *domain invariance* exists during the course of learning.

Finally, because the feature extractor is shared by both the label and domain classifiers, it will learn domain invariant features, as such, the label classifier will be used to classify samples from both the source domain D_S and the target domain D_T.

3 Technologies

3.1 Python

Python, a high-level programming language, is used to write this projects code. Its lack of complexity helps with code readability, quick developing, rapid prototyping and easy testing of desired programs, all of which adds up to a fast learning curve for beginner programmers as well as experienced ones. Although all of the previous reasons are a part of why Python is used in this project, the most important one is its huge amount of available inbuilt libraries, many of which can be used for Machine Learning and mathematical operations. Some of these Machine Learning libraries are TensorFlow and Keras, both of which are used in this project.

3.1.1 TensorFlow Library

TensorFlow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and Deep Learning and the flexible numerical computation core is used across many other scientific areas.

3.1.2 Keras Library

Keras is a high-level neural networks API, written in Python and capable of running on top of the TensorFlow library, among others. It was designed with four guiding principles in mind, user friendliness, modularity, easy extensibility and to work with Python.

Keras also provides an easy way to build and test models called the Sequential API, which is a linear stack of Neural Network layers. Although, due to the complexity of the DANN, we will not be able to use this API, and will opt for the Functional API, which

one of the things it allows is the creation of more complex models with shared layers but different outputs. Additionally, Keras also facilitates the training for our kind of Neural Network with a method to train on batches.

3.2 Google Colaboratory

Google Colaboratory is a free-use research tool for education and machine learning exploration. It is a Jupyter notepad environment that can be used without configuration.

All Collaboratory notebooks are stored in Google Drive, so they are saved and accessible from the cloud.

It is compatible with Python 3.6, and you can use all the language's libraries including TensorFlow and Keras. In addition, it allows execution in two environments, with the CPU or GPU. The GPU runtime environment makes available a size of up to 25GB VRAM, of which at least 12GB VRAM available.

4 Methodology

This chapter is divided in three steps:

- ***Data and Tools Preparation Stage:*** The first step is dedicated to the explanation of the datasets to be used by the learning algorithms as well as what tools are necessary for the completion of the task at hand.

- ***Architecture Building Stage:*** The second step deals entirely with the justification of the requirements posed on the two models used and their design.

- ***Experimentation Stage:*** The third step makes understandable the type of experiments that will be carried out and how they must be in order to obtain robust and high-standard results.

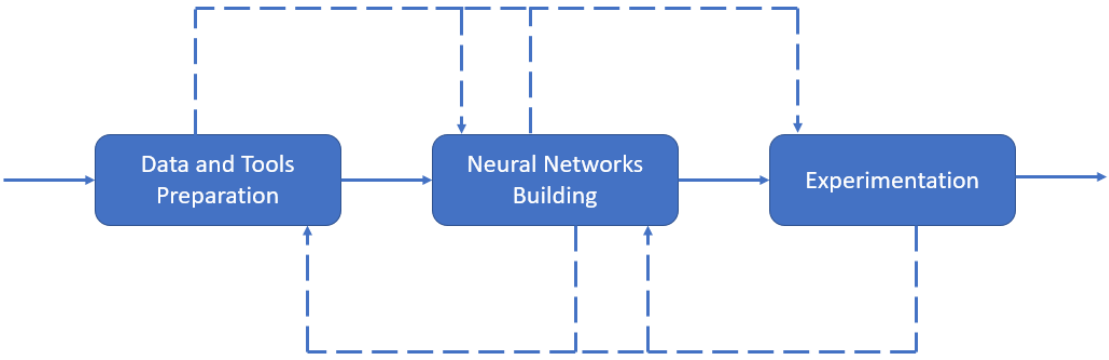

Figure 4.1: Methodology workflow diagram.

These steps are followed to successfully bring about the completion of the implementation and the experiments. The methodology workflow is depicted in Fig. 4.1; while it seems that the steps are supposed to be performed in order, modifications or new additions were added as the project evolved – this means that the steps are not locked by completion of the previous ones.

4.1 Data and Tools Preparation Stage

4.1.1 Data Preparation

There are two different datasets utilized in this project, MNIST and MNM. The first has less importance than the second, as it is too simple, because of this, the experiments will be mainly performed on the second dataset.

The preparation of the data includes the download of the datasets, the creation of target domains and structuring the datasets into training and testing.

- ***MNIST Dataset.*** The MNIST dataset (Modified National Institute of Standards and Technology) is a large dataset of black and white, normalized, 28x28 pixel images containing one handwritten digit from zero to nine.

  The dataset is comprised originally of 65000 images with their corresponding labels. From these, 55000 are used for training, while the other 10000 are used for testing. This initial data will be known as the Default or Source MNIST dataset.

  As we need a target domain for the purpose of the project, advantage is taken over the type of images in Source MNIST. The images are black and white, in particular: the background is black and the digit is white; it is enough to switch the colors to random ones in order to obtain a different domain, but in order to increase the difficulty given that this dataset is simple, the colors are replaced with random areas of the images found in the Berkeley Segmentation Dataset.

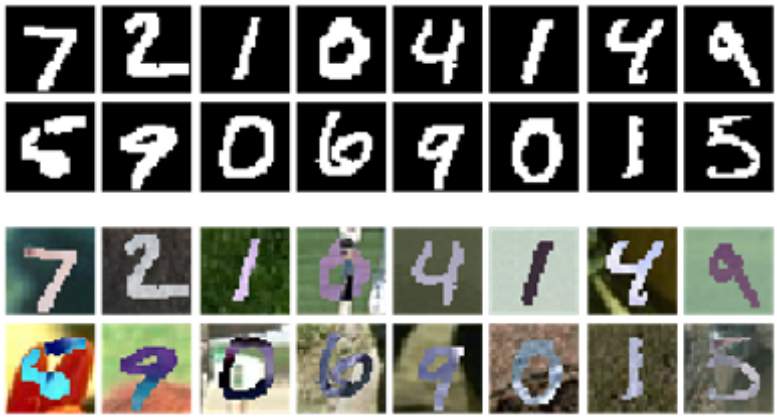

Figure 4.2: Example of images taken from Source MNIST (black and white images) and their respectives from Target MNIST.

Domain	Type	Total Images
Source MNIST	Default	65000
Target MNIST	Synthetic	65000

Table 4.1: Description of the two MNIST datasets used.

- ***MNM Dataset.*** The MNM dataset (Mensural Notation Manuscript) is a collection of five distinct manuscripts from different domains, given that all of them vary between each other because one or more graphical particularities, and contain 40x40 pixel images of music symbols.

  Given the artistic scope of the dataset's domain, it is difficult to indicate exactly what differentiates manuscripts among themselves, but we can roughly add it up to a number of these factors: authors' handwriting style, engraving mechanism, type and color of the paper/parchment, color of the ink, and an amount of possible deterioration, among others.

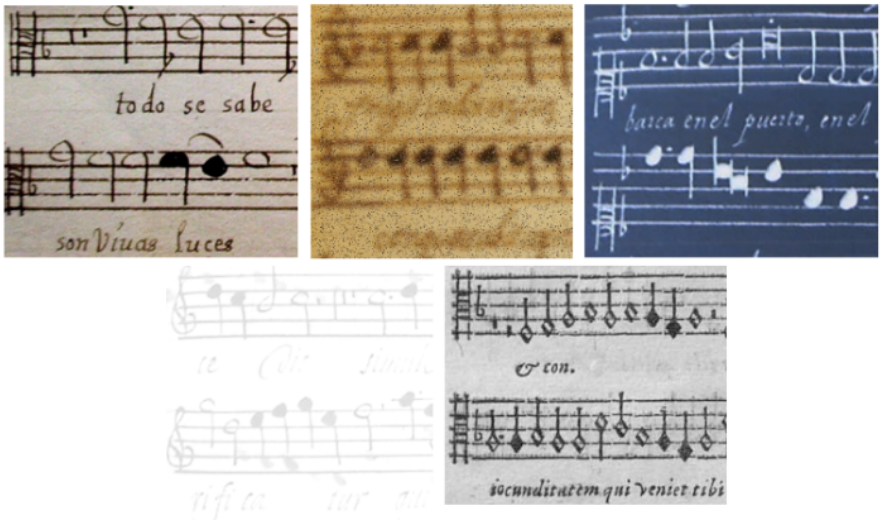

Figure 4.3: Examples taken from every manuscript of MNM. From left to right: b-3-28, b-50-747, b-53-781, b-59-850, BNE-BDH.

  For the most part, the previous factors are present in all of the manuscripts, creating, like this, five domains as seen in Fig. 4.3. Firstly, data from domains b-3-28 through b-59-850 have been obtained from handwritten manuscripts, while BNE-BDH has been typewritten. Secondly, the manuscripts pose varied characteristics, for example, b-50-747 has a lower resolution so it is more blurry, and b-59-850 has an overexposure or very high intensity of light. Thirdly, manuscripts b-3-28, b-50-747, b-59-850 and BNE-BDH have not been altered by adding any type of

additional characteristic, but a distinct type of characteristic was added to b-53-781, with the aim of increasing the difficulty and unpredictability of the state in which future manuscripts can be provided, as the DANN model must be robust to these types of alterations and must not require additional manual labor, e.g. image preprocessing, in order to learn and transfer its knowledge. Finally, a synthetic characteristic, which inverts the colors, was manually applied to manuscript b-53-781, which anyway could represent a different scanner mechanism.

Table 4.2 shows a summary of the datasets used, as well as their type and number of samples they contain. The initial amount of data (column "Total symbols") must be processed, it contains some labels that are not common in all datasets, because not all manuscripts use the same symbols, and when a symbol is used by all it may only be used a few times.

To solve this, the symbol category label sets from each domain have been all intersected in order to obtain a list with only the categories common to all manuscripts, reducing the amount to 15. Additionally, categories which add up to less than 15 elements in all manuscripts have been removed as well, further reducing the amount of categories to 8. This brings us to the final amount of data that will be used for experimentation (see column "Filtered symbols").

Domain	Type	Total Symbols	Filtered Symbols
b-3-28	Handwritten	2581	1760
b-50-747	Handwritten	3191	2587
b-53-781	Handwritten	825	570
b-59-850	Handwritten	8467	5225
BNE-BDH	Typewritten	1356	1199

Table 4.2: Description of the five MNM datasets used.

For the two datasets, two different sets are required for successfully training the networks used. The first, the *training set* $\mathcal{T}$, is the conventional set created from the sample S (Source Domain), which contains input and label pairs $(x_i,\ y_i)$. The second, the *domain set* $\mathcal{D}$, is comprised of data from the input samples S and T (Target Domain), along with its domain-label pairs $(x_i,\ d_i)$, where d_i indicates from which domain it originates (e.g., 0 indicates that it comes from the Source Domain and 1 indicates that it comes from the Target Domain).

4.1.2 Tools Preparation

- ***Keras Library.*** The Keras library is used for building, training and evaluating the models used in this project.

- ***Keras Functional API.*** The Keras functional API is used for defining complex models, such as multi-output models, directed acyclic graphs, models with shared layers, among others.

  The reason this tool is used is for this added flexibility and resiliency when it comes to designing and building models with multiple outputs and shared layers.

- ***TensorFlow Library.*** The TensorFlow library is used for gaining access to the gradient of the architectures in order to carry out the gradient reversal operations mentioned in the State of the Art chapter.

- ***Miscellaneous Libraries.*** Other libraries are used, such as matplotlib, numpy, glob, cv2, json, tarfile, os, pickle, skimage and collections.

  These are needed for image visualization and editing, file writing and reading, and operations on arrays, among others.

4.2 Architecture Building Stage

Because the purpose of the project is to attempt the transfer of knowledge of a NN from one domain to a different one, we will need to build two different networks in order to quantify the transfer and compare results between the two. This has been done for both datasets, given that the two datasets vary significantly between each other, and a more complex model is needed for the MNM dataset, compared to the simpler MNIST model.

Additionally, because of the use of the Keras Functional API, we are able to build networks that have a shared Feature Extractor and then bifurcate after it into a Label Classifier and a Domain Classifier, as seen in Fig. 2.4. This allows for the design, creation and training of a network that can be used both as a CNN model (the Domain Classifier can be ignored during training and then the whole network works just as a CNN approach) and a DANN model. This can be thought of as Object Oriented Programming, different instances of this "master" network are created and they work as independent objects (models) with their own unique attributes (weights).

This means that it is sufficient to create two "master" networks for each of the datasets,

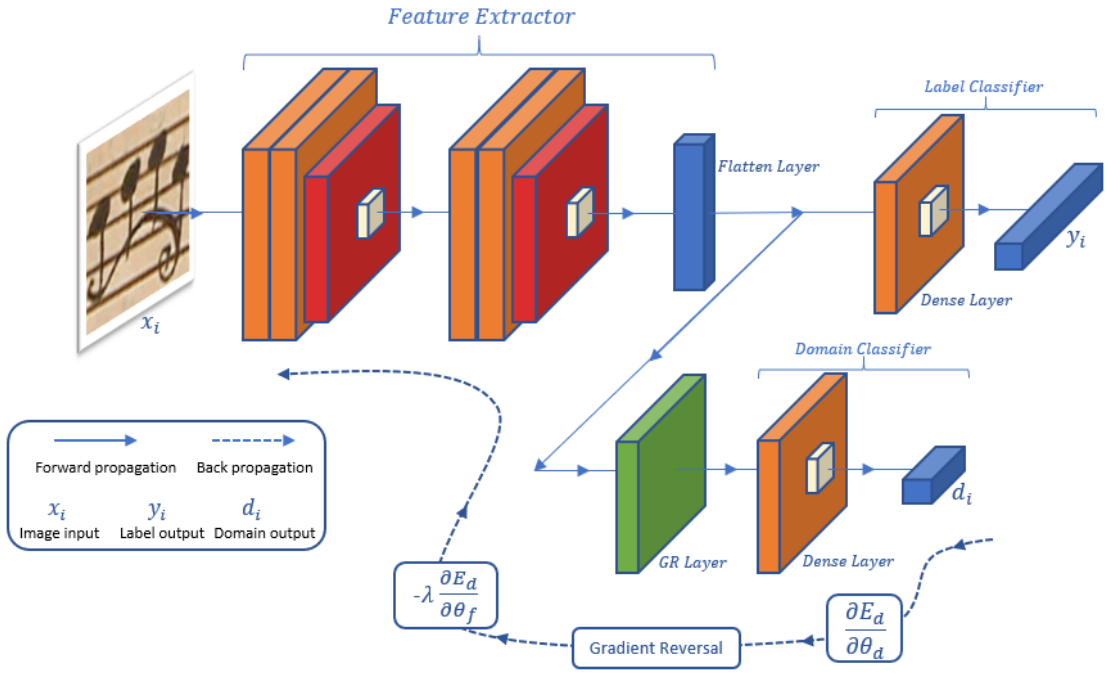

Figure 4.4: The considered architecture for MNM. The Gradient Reversal Layer (GR-Layer), is situated in the Domain Classifier structure.

as observed in Fig. 4.4. Although this figure shows the network for the MNM dataset network, the MNIST one does not differ in design, only in amount of hidden layers and network hyperparameters. Both "master" networks have a Feature Extractor, which is the common part of both models, followed by a bifurcation into the Label Classifier and the Domain Classifier. It must be noted as well, that this Domain Classifier is the one that enables the modification of the gradient during backpropagation via a Gradient Reversal Layer present in it, this is done only if we wish to use the DANN model; as we said, if we only want the CNN approach this classifier is ignored.

4.2.1 Training

Both models, CNN and DANN, are built for the same task, image classification. But, given that they vary in architecture and are designed with a specific purpose in mind, they will be trained very differently from one another. Nonetheless, this can be generalized for both dataset architectures and for both models.

The first model, the CNN approach, will follow a common image classification training approach. This model will be trained on pairs from the *training set* $\mathcal{T}$, and then tested

on the Source and Target Domains.

The second model, the DANN approach, must be trained in a peculiar way. Given that the model makes use of two specific classifiers which share the weights of a part of the network (of the Feature Extractor), a problem arises during its training. If it is trained in a conventional manner just like a CNN model, after fully training one classifier, its knowledge might be lost or become invalid after doing the same to the second classifier. To solve this, the model will make use of a form of pseudo-concurrent training, using small, equally-sized batches obtained from the domain set $\mathcal{D}$ to train one and then the other (e.g., an epoch is comprised of X images, and, using batches of size b, it will be trained X/b times for each epoch). The label classifier will make use of the *training set* $\mathcal{T}$, while the domain classifier will use the *domain set* $\mathcal{D}$. Just as the CNN approach, after training it will then be tested on the Source and Target Domains.

4.3 Experimentation Stage

The results of the Experimentation Stage have been obtained differently for both of the datasets. Because the MNIST dataset is very simple, the results will be obtained by just training as it was previously mentioned.

For the MNM dataset, experimentation followed a meticulous process with different iterations, where one iteration means all possible permutations by electing each manuscript once as the Source Domain and the others as the Target Domain. Additionally, these training iterations were carried out using different values for parameters such as the pseudo-concurrency batch size, values of λ of the gradient reversal layer, the label classifier's learning rate and the domain classifier's learning rate.

Parameter	Values
Batch size	{16, 32, 64, 128, 256, 512}
λ	{0.5, 1.0, 1.5, 2.0}
Classifier Learning Rates	{0.5, 1.0, 1.5}

Table 4.3: Set of values used for the training parameters.

Additionally, the MNM experimentation has been carried out in two stages, the first stage studies how the general tendency varies by doing a complete search on the previously mentioned parameters, the batch size, gradient reversal layer's λ, and learning rates, using the values shown in Table 4.3. Both the CNN and DANN models will use the batch size and the classifier learning rates during the training of the respective models. The λ parameter only affects the DANN model. The second stage of experimentation

carries out tests across all possible manuscript permutations of source-domain using in turn every possible permutation of the parameters.

5 Implementation

The Implementation Chapter, this means the realization through programming of the technical specifications, algorithms and functions required to carry out the experimentation stage, will be divided in the same way as the Methodology Chapter:

- *Data and Tools Implementation.*
- *Architecture Implementation.*
- *Experimentation Implementation.*

5.1 Data and Tools Implementation

5.1.1 Synthetic MNIST Creation

As it was previously mentioned, given that the images in Source MNIST are of a white digit on black background, the Target MNIST will be created by switching the black and white colors each by a random part of images from the Berkeley Segmentation Dataset. This ensures a domain shift between the two MNIST domains.

The important functions used are as follows:

```
def compose_image(digit, background):
  w, h, _ = background.shape
  dw, dh, _ = digit.shape
  x = np.random.randint(0, w - dw)
  y = np.random.randint(0, h - dh)
  bg = background[x:x+dw, y:y+dh]
  return np.abs(bg - digit).astype(np.uint8)

def mnist_to_img(x):
  x = (x > 0).astype(np.float32)
  d = x.reshape([28, 28, 1]) * 255
  return np.concatenate([d, d, d], 2)

```

```
def create_mnistm(X, background_data):
  X_ = np.zeros([X.shape[0], 28, 28, 3], np.uint8)
  for i in range(X.shape[0]):
    bg_img = rand.choice(background_data)
    d = mnist_to_img(X[i])
    d = compose_image(d, bg_img)
    X_[i] = d
  return X_
```

Listing 5.1: Synthetic MNIST creation functions.

The process of creating new synthetic images is controlled by the *create_mnistm* function. In its *for* loop, the function iterates over the images of the Source MNIST dataset; for each iteration it randomly picks a background image from the Berkeley Segmentation Dataset, it then decomposes into a numpy array the source image and finally it assembles the new target (synthetic) image and saves it.

The *mnist_to_img* function simply decomposes an image into a numpy array so parts of it can be edited in the *compose_image* function. This last function is the one tasked with switching the black backgrounds and white digits of the Source MNIST with random parts of random images from the Berkeley Segmentation Dataset.

5.1.2 MNM Filters

The filers applied to the MNM database, mentioned in the Methodology chapter, followed this process: first, remove symbol categories not common to all five manuscripts used, and second, remove symbol categories that do not add up to 15 in all manuscripts.

The first part of the filters makes use of the original X and Y sets (input and label sets) and an intersected list of the labels of all manuscripts, it is carried out by the following function:

```
def __filter_data(origin_X, origin_Y, intersected_Y):
    new_X = []
    new_Y = []

    for x, y in zip(origin_X, origin_Y):
        if y in intersected_Y:
            new_X.append(x)
            new_Y.append(intersected_Y.index(y))

    return np.array(new_X), np.array(new_Y)
```

Listing 5.2: Non-common symbol category removal function.

It iterates on the X and Y sets, and for each iteration checks if the symbol category (Y) is in the intersected list. If it is found in it, then this X and Y pair are stored in the new filtered sets. If it is not found, then it is simply ignored.

The second part of the filters appends in an empty array those labels that do not count up to a specified amount (in the experiments, up to 15, set by the *MINSIZE* variable). Afterwards, the labels in this array are removed from the intersected label list used in the first filter, and the same function as above, *__filter_data*, is called in order to create the new X and Y sets without the low count categories.

```
MINSIZE = 15
to_elim = []

for i in range(len(datasets)):
    counter = collections.Counter(datasets[i][1])
    for y in intersected_Y:
        d = counter.get(intersected_Y.index(y))
        if d < MINSIZE and y not in to_elim:
            to_elim.append(y)

for y in to_elim:
    intersected_Y.remove(y)
```

Listing 5.3: Symbol category count filter.

They are not coded as one function so as to maintain a separation of the two filters if the need arises to not use one or both of them.

5.2 Architecture Implementation

Because there are two "master" architectures, one for the MNIST database and one for the MNM database, they will be explained separately. But both architectures contain a CNN and a DANN model, and follow the same design as seen in Fig. 4.4.

Let $Conv2D(f, (k_1, k_2))$ be a convolutional layer with f filters and kernel size of $k_1 \times k_2$, $MaxPooling((p_1, p_2))$ be a max-pooling operation layer with pool size $p_1 \times p_2$, $Dense(u, a)$ be a dense layer with u units and activation a, and $Dropout(d)$ be a dropout operation layer with a dropout ratio of d.

5.2.1 Gradient Reversal Layer

It must also be noted that the GRLayer() is the Gradient Reversal Layer, the layer tasked with the modification of the gradient during backpropagation. It is programmed in such a way so that it can be called wherever we want it as if it was a normal Keras layer; it works by calling the GradientReversal method with the desired lambda value, the value by which we want to modify the gradient in each backpropagation pass.

```
1  def reverse_gradient(X, hp_lambda):
2    ...
3
4    @tf.RegisterGradient(grad_name)
5    def _flip_gradients(op, grad):
6        return [tf.negative(grad) * hp_lambda]
7
8    g = K.get_session().graph
9    with g.gradient_override_map({'Identity': grad_name}):
10       y = tf.identity(X)
11
12   return y
```

Listing 5.4: Gradient Reversal Layer code snippet in Python.

The layer has additional code, including the call function, attributes, etc. But the most important part is the one seen in the snippet.

The decorator *@tf.RegisterGradient(grad_name)* registers the gradient function defined by the function in lines *5* and *6*. Then, in lines *9* and *10*, we use *g.gradient_override_map* to change the default gradient by the one registered in the decorator. Like this, we achieve the gradient reversal operation we seek.

This code uses the TensorFlow library, as the according functions are not available yet in Keras.

Assuming there is previous correct code, an example of a call to use the layer:

```
1  Flip = GradientReversal(1.0)
2  domain_classifier = Flip(model)
```

Listing 5.5: Gradient Reversal Layer example.

5.2.2 MNIST Architecture

The architecture receives an input image of size 28×28 pixels and transfers it to the Feature Extractor block. The Feature Extractor is composed by two 2D convolutional layers, a max-pooling layer, a dropout layer and ends with a Flatten layer that converts the output of the Feature Extractor into a vector. Then it follows a bifurcation that leads separately to the Label Classifier and the Domain Classifier.

```
def mnist_model(input_shape, label_shape, domain_shape):
  inputs = Input(shape=input_shape)
  feature_extractor = Conv2D(32, (3, 3), activation='relu', input_shape=
    input_shape)(inputs)
  feature_extractor = Conv2D(64, (3, 3), activation='relu')(
    feature_extractor)
  feature_extractor = MaxPooling2D(pool_size=(2, 2))(feature_extractor)
  feature_extractor = Dropout(0.25)(feature_extractor)
  feature_extractor = Flatten()(feature_extractor)

  label_classifier = Dense(128, activation='relu')(feature_extractor)
  label_classifier = Dropout(0.5)(label_classifier)
  label_classifier_outputs = Dense(label_shape, activation='softmax')(
    label_classifier)

  Flip = GradientReversal(1.0)
  domain_classifier = Flip(feature_extractor)
  domain_classifier = Dense(128, activation='relu')(domain_classifier)
  domain_classifier = Dropout(0.5)(domain_classifier)
  domain_classifier_outputs = Dense(domain_shape, activation='softmax')(
    domain_classifier)

  label_classifier = Model(inputs=inputs, outputs=
    label_classifier_outputs)
  domain_classifier = Model(inputs=inputs, outputs=
    domain_classifier_outputs)

  return label_classifier, domain_classifier
```

Listing 5.6: MNIST architecture using Keras.

- The Feature Extractor considers one convolutional block, set as: $Conv2D(32, (3, 3)) \rightarrow Conv2D(64, (3, 3)) \rightarrow MaxPooling((2, 2)) \rightarrow Dropout(0.25)$.
- The Label Classifier is built as: $Dense(128, ReLU) \rightarrow Dropout(0.5) \rightarrow Dense(10, softmax)$, with output being the number of digits.
- The Domain Classifier consists of: $GRLayer() \rightarrow Dense(128, ReLU) \rightarrow Dropout(0.5) \rightarrow Dense(2, softmax)$, with output being two domains, $d_i \in \{0, 1\}$.

5.2.3 MNM Architecture

The architecture receives an input image of size 40×40 pixels and transfers it to the Feature Extractor block. The Feature Extractor is composed by two blocks of two 2D convolutional layers, a max-pooling layer, a dropout layer and ends with a Flatten layer that converts the output of the Feature Extractor into a vector. Then it follows a bifurcation that leads separately to the Label Classifier and the Domain Classifier.

```
def mensural_model(input_shape, label_shape, domain_shape):
  inputs = Input(shape=input_shape)
  feature_extractor = Conv2D(32, (3, 3), padding='same', activation='relu
    ', input_shape=input_shape)(inputs)
  feature_extractor = Conv2D(64, (3, 3), activation='relu')(
    feature_extractor)
  feature_extractor = MaxPooling2D(pool_size=(2, 2))(feature_extractor)
  feature_extractor = Dropout(0.25)(feature_extractor)

  feature_extractor = Conv2D(64, (3, 3), padding='same', activation='relu
    ')(feature_extractor)
  feature_extractor = Conv2D(64, (3, 3), activation='relu')(
    feature_extractor)
  feature_extractor = MaxPooling2D(pool_size=(2, 2))(feature_extractor)
  feature_extractor = Dropout(0.3)(feature_extractor)
  feature_extractor = Flatten()(feature_extractor)

  label_classifier = Dense(128, activation='relu')(feature_extractor)
  label_classifier = Dropout(0.5)(label_classifier)
  label_clasifier_outputs = Dense(label_shape, activation='softmax')(
    label_classifier)

  Flip = GradientReversal(1.0)
  domain_classifier = Flip(feature_extractor)
  domain_classifier = Dense(128, activation='relu')(domain_classifier)
  domain_classifier = Dropout(0.5)(domain_classifier)
  domain_classifier_outputs = Dense(domain_shape, activation='softmax')(
    domain_classifier)

  label_classifier = Model(inputs=inputs, outputs=
    label_classifier_outputs)
  domain_classifier = Model(inputs=inputs, outputs=
    domain_classifier_outputs)

  return label_classifier, domain_classifier
```

Listing 5.7: MNM architecture using Keras.

- The Feature Extractor considers two convolutional blocks, set as:
  $Conv2D(32,(3,3)) \rightarrow Conv2D(64,(3,3)) \rightarrow MaxPooling((2,2)) \rightarrow Dropout(0.25) \rightarrow Conv2D(64,(3,3)) \rightarrow Conv2D(64,(3,3)) \rightarrow MaxPooling((2,2)) \rightarrow Dropout(0.3)$.

- The Label Classifier is built as: $Dense(128, ReLU) \rightarrow Dropout(0.5) \rightarrow Dense(18, softmax)$, with output being the filtered number of symbol categories.

- The Domain Classifier consists of: $GRLayer() \rightarrow Dense(128, ReLU) \rightarrow Dropout(0.5) \rightarrow Dense(2, softmax)$, with output being two domains, $d_i \in \{0, 1\}$.

5.3 Experimentation Implementation

The experiments were conducted using a generalized code in order to train the architectures for both databases without the need to write additional, duplicate, code.

The first thing the algorithm will do is create the models according to the database used. According to the *db* flag, the models created will vary in input and output, as well as datasets used.

```
...
if args.db == 'mnist':
    model_label, model_domain = utilModels.mnist_model(input_shape,
                                num_labels, num_domains, args)
elif args.db == 'mensural':
    model_label, model_domain = utilModels.mensural_model(input_shape,
                                num_labels, num_domains, args)
...
```

Listing 5.8: Database flag check.

As there are two types of models in each of the architectures, the CNN and DANN models, the code will check the *mode* flag. If the flag is *"dann"*, then it will train the DANN model using the *train_dann* created function, not found in any library. If the flag is *"cnn"*, the CNN model will be trained, using the *fit* function found in the Keras library.

```
if args.mode == "dann":
    ...
    train_dann(model_label, model_domain,
                            datasets[i]['x_train'],
                            datasets[i]['y_train'],
                            datasets[j]['x_train'],
                            datasets[j]['y_train'],
                            weights, args)
    ...
elif args.mode == "cnn":
    ...
    model_label.fit(datasets[i]['x_train'], datasets[i]['y_train'],
                            batch_size=args.batch,
```

```
                                epochs=args.epochs,
                                verbose=2,
                                shuffle=True,
                                validation_data=(datasets[j]['x_train'],
                                datasets[j]['y_train']))
    ...
```

Listing 5.9: Model flag check.

The *fit* Keras function simply trains the model for a given number of epochs, using additional parameters that can also be used. The *train_dann* function (seen in Listing 5.10), on the other hand is more complicated than just a straightforward training function, given that the DANN model requires a unique pseudo-concurrent training method.

Firstly, the function creates two batch generators, for the Source and Target Domains each. These generators will be used in order to obtain batches of training data from each domain. These batches can be considered arrays of pairs, where the first element is the same input x_i from X; but the second element is this input's image label y_i from Y_L for the Source Domain batch generator, and this input's domain label d_i from Y_D for the Target Domain batch generator.

Secondly, local variables are set with the configuration parameters needed to emulate the integrated loop of the *fit* Keras function. These variables are: *imgs_per_epoch*, that indicates how many images constitute an epoch (a training iteration); *e*, which counts the elapsed epochs; *img_nr*, which counts how many images of the current epoch have been trained on; *best_label_acc*, which stores the best label classifier accuracy, in order to save the best model during training.

After this initial setup, the training loop begins. The loop's break point is when the current epoch passes the specified maximum number of epochs set in the configuration. While it has not finished, it will do the following:

- Obtain two batches, of Source and Target Domain data.
- Train both classifiers using their corresponding batches via the *train_on_batch* Keras function. The label classifier will train on the Source Domain batch, while the domain classifier on the Target Domain batch.
- Increase the counter of images trained on, during the current epoch, by the batch size.
- If the amount of images trained on hasn't reached the maximum amount per epoch, then continue with the loop iterations.

- If the amount of images trained on has reached this maximum, then the algorithm resets this counter to zero, increases by one the epoch counter and then continues by checking if the accuracy of the label classifier has improved. If it is improved, then the weights of the label classifier are saved over any previous ones, if not then the loop continues.

```
def train_dann(model_label, model_domain, source_x_train, source_y_train,
               target_x_train, target_y_train, weights, config):

    gen_source_batch = util.batch_generator([source_x_train,
    source_y_train], config.batch)
    gen_target_batch = util.batch_generator([target_x_train,
    target_y_train], config.batch // 2)

    imgs_per_epoch = source_x_train.shape[0]
    e = 0
    img_nr = 0
    best_label_acc = 0

    while e < config.epochs:
        X_s, y_s = next(gen_source_batch)
        X_t, __ = next(gen_target_batch)

        X_d = np.vstack([X_s[0:config.batch // 2], X_t])
        y_d = np.vstack([np.tile([1., 0.], [config.batch // 2, 1]),
              np.tile([0., 1.], [config.batch // 2, 1])])
        X_d, y_d = shuffle(X_d, y_d)

        label_loss, label_acc = model_label.train_on_batch(X_s, y_s)
        domain_loss, domain_acc = model_domain.train_on_batch(X_d, y_d)

        img_nr += config.batch
        saved = ""
        if img_nr > imgs_per_epoch:
            img_nr = 0
            e += 1
            if best_label_acc < label_acc:
                best_label_acc = label_acc
                model_label.save(weights)
                saved = "SAVED"
```

Listing 5.10: DANN training algorithm.

6 Experimentation

The experiments are divided in two main groups, MNIST and MNM experimentation; and given that the MNIST database is simple, this chapter will dive more into the MNM database experiments. Nonetheless, a small section is dedicated to MNIST.

Additionally, in the tables showing experimentation results, we name "D_s Acc." and "D_t Acc." as the average source and target domain accuracy in percentage, and all experiments were obtained using 100 epochs.

6.1 MNIST Experimentation

MNIST experimentation in Table 6.1 has been carried out with a batch size of 64. Both models have been used, in order to obtain comparisons between the two; additionally, the models first trained on the Default MNIST and then tested on the Synthetic MNIST, and afterwards, the other way around.

As seen in Fig. 6.1, for training on Default MNIST, the DANN model obtains a target domain accuracy of 67.82%, which is a 16.14% increase from the CNN's 51.68%. On the other hand, for training on Synthetic MNIST the target domain accuracy decreases for the DANN model, this may be because of the simplicity of Default MNIST or because the domains are very similar.

Domain		CNN		DANN	
Source	Target	D_s Acc.	D_t Acc.	D_s Acc.	D_t Acc.
Default MNIST	Synthetic MNIST	97.98	51.68	95.40	67.82
Synthetic MNIST	Default MNIST	94.95	97.08	92.18	94.63

Table 6.1: Results for MNIST database with CNN and DANN architectures.

Other experiments were also conducted using a batch size of 32, Table 6.2. Results worsened for this batch, this can be because for a low amount of inputs, the weights are changed too slowly during training.

Domain		DANN	
Source	Target	D_s Acc.	D_t Acc.
Default MNIST	Synthetic MNIST	95.18	66.15
Synthetic MNIST	Default MNIST	89.55	90.48

Table 6.2: Results for MNIST database with DANN architecture.

6.2 MNM Experimentation

The experimentation of the MNM database has been carried out in two stages, the first stage studies how the general tendency varies by doing a complete search on the previously mentioned parameters, the batch size, gradient reversal layer's λ, and learning rates, using the values shown in Table 4.3. The second stage of experimentation carries out tests across all possible manuscript permutations of source-domain using in turn every possible permutation of the parameters.

Results obtained for the first stage are shown in separated tables, where, for each possible value of the parameter at issue, the average source and target domain accuracies for the CNN and DANN models are shown, when possible. These results are the average obtained as a consequence of carrying out experiments across all possible manuscript permutations of source-target domains, using in turn every possible permutation of the considered parameters. This means that a total of 8640 experiments with different configurations have been carried out: 20 possible manuscript combinations $\times$ 2 different models (CNN and DANN) $\times$ 6 batch sizes $\times$ 4 λ values $\times$ 3 label classifier learning rate $\times$ 3 domain classifier learning rate. These results will allow us to get an idea of how the parameters affect the general trend of the results.

The tendency for the batch size, Table 6.3, is for results to decrease substantially as the batch size increases. This fulfills what was previously mentioned about the need to use fairly small sizes, as training one classifier with a large batch will undermine the training of the other. The same tendency arises if we use batches that are too small, since in this case the weights are updated using very few samples. The best results tend to originate from a batch size of 64 samples.

The λ parameter (see Table 6.4) is the variable by which the Gradient Reversal Layer multiplies the derivative of the domain classifier in order for the label classifier to subsequently be trained invariantly to domains. This parameter is equivalent to the learning rate parameter of the back-propagation algorithm, but applied to the learning of the domain invariant characteristics by the Feature Extractor block. Note that this is a particular parameter to the DANN model, and so only its results can be reported. They

Batch	CNN		DANN	
	D_s Acc.	D_t Acc.	D_s Acc.	D_t Acc.
16	96.06	18.25	72.20	37.67
32	96.39	18.89	77.60	39.11
64	96.39	18.46	81.30	39.74
128	96.25	18.67	82.10	39.61
256	96.00	19.36	82.78	36.73
512	95.13	19.48	80.15	35.88

Table 6.3: Batch sizes influence in CNN and DANN architectures.

show a general tendency to obtain better results for small values of λ, so in this case it is better to use a small factor to learn the domain invariant characteristics. This may be due to the fact that if a high value is used, the characteristics shared by the two networks are more adjusted for domain detection, spoiling the result obtained by the label classifier (and undoing what it had already learned). And therefore, it is better to adjust the weights little by little in each iteration, to reach a balance between the two networks.

λ	DANN	
	D_s Acc.	D_t Acc.
-0.5	86.77	39.53
-1.0	80.30	38.62
-1.5	76.57	37.24
-2.0	73.83	37.10

Table 6.4: Influence of λ values in the performance of the DANN architecture.

In Tables 6.5 and 6.6, the results obtained by varying the learning rates of the two networks are analyzed. The general tendency for the best learning rate values varies between the two classifiers. Table 6.5 shows the influence of the label classifier's learning rate in the performance of the CNN and DANN architecture. In this case, having a learning rate of 0.5 for the label classifier obtains the best results. Table 6.6 shows the influence of the domain classifier's learning rate in the performance of the DANN architecture. In this other case, having a learning rate of 1.0 for the domain classifier obtains the best results. Both classifiers obtain bad results for a high value of their respective learning rate. As previously argued, this can be motivated by the fact that having a shared weight section, it is better to modify the weights little by little in each iteration. Therefore, it seems that, on average, it is better to use a domain learning rate

value slightly higher than that used for the labels.

LR1	CNN		DANN	
	D_s Acc.	D_t Acc.	D_s Acc.	D_t Acc.
0.5	95.58	20.33	88.36	39.76
1.0	96.21	18.97	79.49	38.89
1.5	96.32	17.27	70.11	35.69

Table 6.5: Influence of the label classifier's learning rate in the performance of the CNN and DANN architecture.

LR2	DANN	
	D_s Acc.	D_t Acc.
0.5	79.43	37.81
1.0	79.39	38.70
1.5	39.27	37.86

Table 6.6: Influence of the domain classifier's learning rate in the performance of the DANN architecture.

The results for the second stage of the experimentation are shown in Table 6.7. It reports the results obtained by the CNN and DANN networks for all possible combinations of the five datasets used. In addition, the "target diff" column is added, which shows the difference between the accuracy for the target domain obtained by the DANN and the CNN.

Results for the typewritten manuscript (BNE-BDH) as source and target are promising, as they have good increases in DANN accuracies compared to the CNN ones. An anomaly occurs with b-3-28 and b-53-781 as source domains and BNE-BDH as target, where we obtained results that do not outperform the CNN classifier, given that the target differences are 3.34% and 0.42% respectively. This may be due to the number of samples in the source dataset, since they represent the two datasets with fewer samples.

It is also observed that the pair b-3-28 and b-50-747 obtains poor results in the two possible combinations (source, target), with the following target accuracy difference: 5.79% (b-3-28 as source) and -7.10% (b-50-747 as source). In this case, these worse results are probably caused by the similarity of the domains. In Figure 4.3, one can see how these two domains are those that present the most similar writing, only with differences in the overall color of the image. In general, it has been observed that DANN architecture requires the source and target domains to have significant differences, since,

if this is not the case, it forces the label classifier to be trained with domain invariance, which results in worse or similar accuracies as the CNN architecture. Note that, if the domains are similar, the DANN is actually modifying the features that are already suitable for both.

Additionally, the opposite happens when there is too much difference between domains, as can be seen whenever b-53-781 is used as the target domain (see Fig. 4.3). Using b-50-747 and b-59-850 as sources, the DANN architecture obtains the experiments' maximum target accuracy difference: 84.21%.

As a summary, the proposed DANN architecture obtains on average 67.45% target accuracy, which is a 37.80% increase on average than the CNN architecture.

Source	Target	CNN		DANN		Target Diff.
		D_s Acc.	D_t Acc.	D_s Acc.	D_t Acc.	
BNE-BDH	b-3-28	95.83	20.17	88.75	50.28	30.11
	b-50-747	95.83	14.67	92.92	42.08	27.41
	b-53-781	95.83	4.39	90.83	65.79	61.40
	b-59-850	96.25	23.83	90.42	52.82	28.99
b-3-28	BNE-BDH	97.16	37.08	95.17	40.42	3.34
	b-50-747	96.88	73.55	96.02	79.34	5.79
	b-53-781	97.16	16.67	96.02	95.61	78.94
	b-59-850	97.16	53.88	96.02	91.10	37.22
b-50-747	BNE-BDH	97.30	10.42	92.28	32.50	22.08
	b-3-28	97.30	66.48	92.66	59.38	-7.10
	b-53-781	97.68	0.88	94.02	85.09	84.21
	b-59-850	97.49	32.34	95.37	85.84	53.50
b-53-781	BNE-BDH	97.37	35.00	97.37	35.42	0.42
	b-3-28	97.37	8.81	97.37	82.39	73.58
	b-50-747	97.37	4.39	97.37	67.76	60.42
	b-59-850	97.37	23.83	97.37	91.77	64.78
b-59-850	BNE-BDH	99.52	7.50	99.14	29.58	22.08
	b-3-28	99.52	79.55	99.04	85.80	6.25
	b-50-747	99.62	62.17	99.23	80.50	18.34
	b-53-781	99.52	11.40	99.14	95.61	84.21
Average		97.48	29.66	95.33	67.45	37.80

Table 6.7: Best results obtained for the different combinations of the datasets used as source and target domains. Target Diff column shows DANN's D_t accuracy minus CNN's D_t accuracy.

7 Conclusions

This work focuses on the study of the use of Domain Adaptation techniques in the context of image classification. These techniques dictate that a domain invariance must exist during training so classification decisions can be made based on features that are both discriminative for labels, yet invariant to the change of domains. In the project, an existing DANN architecture is implemented; as one of the parts it is comprised of, the domain classifier, includes a gradient reversal layer that will, during backpropagation, ensure these domain invariant features to emerge.

The evaluation of the DANN architecture for the more meaningful database, the Mensural Notation Manuscript database, is carried out by firstly studying how different values of training parameters can affect the general tendency, and then studying the average target domain accuracies using permutations for the five used manuscripts. The parameters evaluated are the training batch size, the gradient reversal layer's λ, and the learning rates for both label and domain classifiers.

The classification performance obtained by the proposed architecture for the target domain generally outperforms a CNN approach, as this implementation results in an average of 67.45%, an increase of 37.80% from the CNN's 29.66%. It should be kept in mind that these classification results (of almost 70% on average), are obtained without using any label from target domain; that is, only adapting the knowledge learned from the source domain. It is also worthwhile mentioning the result obtained for the adaptation between typewritten and handwritten domains, reaching in some cases a 66% of accuracy, 61% better than with a CNN.

Future work includes experiments to improve these results with a greater amount of datasets and domains, increasing the number of labels considered. Also, different strategies are intended to evaluate combining this approach with semi-supervised and incremental methods. Additionally, this research can also be extended to work with the direct detection of the music symbols in the images [7], instead of assuming a previous segmentation.

It must also be noted that an article regarding this subject was submitted to and accepted in: 8th Iberian Conference Pattern Recognition and Image Analysis, 2019. The article in question is: **Tudor N. Mateiu, Antonio-Javier Gallego, and Jorge Calvo-Zaragoza "Domain Adaptation for Handwritten Symbol Recognition:**

A Case of Study in Old Music Manuscripts".

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. pages 102–104. MIT Press, 2016. http://www.deeplearningbook.org.

[2] Jing Zhang, Wanqing Li, and Philip Ogunbona. Transfer learning for cross-dataset recognition: A survey. 2017.

[3] Yaroslav Ganin and Victor S. Lempitsky. Unsupervised domain adaptation by backpropagation. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 1180–1189, 2015.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. page 326. MIT Press, 2016. http://www.deeplearningbook.org.

[5] Jayesh Bapu Ahireh. The artificial neural networks handbook: Part 4. https://dzone.com/articles/the-artificial-neural-networks-handbook-part-4. Accessed: 2019-08-14.

[6] Matt Mazur. A step by step backpropagation example. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/. Accessed: 2019-08-11.

[7] Alexander Pacha and Jorge Calvo-Zaragoza. Optical music recognition in mensural notation with region-based convolutional neural networks. In *19th International Society for Music Information Retrieval Conference*, pages 240–247, Paris, France, 2018.